IS 6733
Deep Learning on Cloud Platforms

Lecture 2a

Python Tutorial - NumPy

Dr. Yuanxiong Guo

Department of Information Systems and Cyber Security

Python Checklist in Machine Learning

- Essential libraries and tools in data science
  - Jupyter Notebook/Colab
    - https://www.youtube.com/watch?v=vVe648dJOdI
  - NumPy
  - Pandas
  - Matplotlib
  - Scikit-Learn
  - Keras/TensorFlow

NumPy and SciPy

- NumPy and SciPy are open-source add-on modules to Python that provide common mathematical and numerical routines in pre-compiled, fast functions.
- These are growing into highly mature packages that provide functionality that meets, or perhaps exceeds, that associated with common commercial software like MATLAB.
- The NumPy (Numeric Python) package provides basic routines for manipulating large arrays and matrices of numeric data.
- The SciPy (Scientific Python) package extends the functionality of NumPy with a substantial collection of useful algorithms, like minimization, Fourier transformation, regression, and other applied mathematical techniques.

NumPy

- import numpy as np
  - This statement will allow us to access NumPy objects using np.X instead of numpy.X.
- The central feature of NumPy is the *array* object class
- Arrays are similar to lists in Python, except that every element of an array must be of the same type (typically a numeric type like *float* or *int*)
- Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists

Array Basics

- An array can be created from a list

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1.,  4.,  5.,  8.])
>>> type(a)
<type 'numpy.ndarray'>
```

- Array elements are accessed, sliced, and manipulated just like lists:

```
>>> a[:2]
array([ 1.,  4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array([ 5.,  4.,  5.,  8.])
```

- Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

Array Basics

- Array Slicing works with multiple dimensions

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:,-2:]
array([[ 5.,  6.]])
```

- The *shape* property of an array returns a tuple with the size of each array dimension, and the *dtype* property tells you what type of values are stored by the array

```
>>> a.shape
(2, 3)
```

```
>>> a.dtype
dtype('float64')
```

- The *len* function returns the length of the first axis, and the *in* statement can be used to test if values are present in an array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> len(a)
2
```

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> 2 in a
True
>>> 0 in a
False
```

Array Basics

- Arrays can be reshaped using tuples that specify new dimensions

```
>>> a = np.array(range(10), float)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a = a.reshape((5, 2))
>>> a
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 8.,  9.]])
>>> a.shape
(5, 2)
```

- Notice that the reshape function creates a new array and does not itself modify the original array.
- Keep in mind that Python's name-binding approach still applies to arrays. The *copy* function can be used to create a new, separate copy of an array in memory if needed:

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

Array Basics

- Lists can also be created from arrays using .tolist()
- Raw data in an array can be converted into/from a binary string using .tostring()/.fromstring()
- One can fill an array with a single value

```
>>> a = array([1, 2, 3], float)
>>> a
array([ 1.,  2.,  3.])
>>> a.fill(0)
>>> a
array([ 0.,  0.,  0.])
```

- Arrays can be transposed by using .transpose()
- One-dimensional versions of multi-dimensional arrays can be generated with .flatten()
- Two or more arrays can be concatenated together using the concatenate function with a tuple of the arrays to be joined.

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

Array Basics

- The dimensionality of an array can be increased using the newaxis constant in bracket notation:

```
>>> a = np.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
>>> a[:,np.newaxis]
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a[:,np.newaxis].shape
(3,1)
>>> b[np.newaxis,:]
array([[ 1.,  2.,  3.]])
>>> b[np.newaxis,:].shape
(1,3)
```

- Notice here that in each case the new array has two dimensions; the one created by newaxis has a length of one.
- The newaxis approach is convenient for generating the proper dimensioned arrays for vector and matrix mathematics.

Array Mathematics

- When standard mathematical operations are used with arrays, they are applied on **element-by-element** basis. That means the arrays should be the same size during addition, subtraction, etc.
- For two-dimensional arrays, multiplication remains elementwise and ***does not*** correspond to matrix multiplication.

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])
```

- For arrays that do not match in **the number of dimensions**, Python will *broadcast* them to perform mathematical operations.

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b
array([-1.,  3.])
>>> a + b
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

Array Mathematics

- In addition to the standard operators, NumPy offers a large library of common mathematical functions that can be applied elementwise to arrays
  - abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh,
  - floor, ceil, rint
  - pi
  - e
- Iterate over arrays

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
...   print x
... <hit return>
1
4
5
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
...   print x
... <hit return>
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for (x, y) in a:
...   print x * y
...  <hit return>
2.0
12.0
30.0
```

For multidimensional arrays, iteration proceeds over the first axis such that each loop returns a subsection of the array:

Multiple assignment can also be used with array iteration:

Basic Array Operations

- Many functions exist for extracting whole-array properties
  - Member functions of the arrays such as *.sum()* or *.prod()*
  - Standalone functions in the NumPy module such as *np.sum()* or *np.prod()*
- A number of routines enable computation of statistical quantities in array datasets
  - .mean(), .var(), .std(), .min(), .max(), .argmin(), .argmax()
- For multidimensional arrays, each of the functions thus far described can take an optional argument *axis* that will perform an operation along only the specified axis

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
>>> a.mean(axis=1)
array([ 1.,  1.,  4.])
>>> a.min(axis=1)
array([ 0., -1.,  3.])
>>> a.max(axis=0)
array([ 3.,  5.])
```

Basic Array Operations

- Like lists, arrays can be sorted and clipped

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1.,  0.,  2.,  5.,  6.])
```

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.clip(0, 5)
array([ 5.,  2.,  5.,  0.,  0.])
```

- Unique elements can be extracted from an array

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1.,  4.,  5.,  7.])
```

- For two dimensional arrays, the diagonal can be extracted

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> a.diagonal()
array([ 1.,  4.])
```

Basic Array Operations

- Comparison Operators and Value Testing

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
```

```
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
```

```
>>> a = np.array([1, 3, 0], float)
>>> a > 2
array([False,  True, False], dtype=bool)
```

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

```
>>> a = np.array([1, 3, 0], float)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> b = np.array([True, False, True], bool)
>>> np.logical_not(b)
array([False,  True, False], dtype=bool)
>>> c = np.array([False, True, False], bool)
>>> np.logical_or(b, c)
array([ True,  True,  True], dtype=bool)
```

```
>>> a = np.array([[0, 1], [3, 0]], float)
>>> a.nonzero()
(array([0, 1]), array([1, 0]))
```

```
>>> a = np.array([1, np.NaN, np.Inf], float)
>>> a
array([  1.,  NaN,  Inf])
>>> np.isnan(a)
array([False,  True, False], dtype=bool)
>>> np.isfinite(a)
array([ True, False, False], dtype=bool)
```

Basic Array Operations

- Array Item Selection and Manipulation

Boolean arrays can be used as array selectors:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False,  True]], dtype=bool)
>>> a[a >= 6]
array([ 6.,  9.])
```

In addition to Boolean selection, it is possible to select using integer arrays. Here, the integer arrays contain the *indices* of the elements to be taken from an array. Consider the following one-dimensional example:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a[b]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

In other words, we take the 0^{th}, 0^{th}, 1^{st}, 3^{rd}, 2^{nd}, and 1^{st} elements of `a`, in that order, when we use `b` to select elements from `a`. Lists can also be used as selection arrays:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> a[[0, 0, 1, 3, 2, 1]]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Basic Array Operations

- Vector and matrix mathematics

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([  6.,  11.])
>>> np.dot(a, b)
array([  3.,  13.])
>>> np.dot(a, c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
>>> a
array([[ 4.,  2.,  0.],
       [ 9.,  3.,  7.],
       [ 1.,  2.,  1.]])
>>> np.linalg.det(a)
-53.999999999999993
```

One can find the eigenvalues and eigenvectors of a matrix:

```
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9.        ,  2.44948974, -2.44948974])
>>> vecs
array([[-0.3538921 , -0.56786837,  0.27843404],
       [-0.88473024,  0.44024287, -0.89787873],
       [-0.30333608,  0.69549388,  0.34101066]])
```

The inverse of a matrix can be found:

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815,  0.07407407, -0.25925926],
       [ 0.2037037 , -0.14814815,  0.51851852],
       [-0.27777778,  0.11111111,  0.11111111]])
>>> np.dot(a, b)
array([[  1.00000000e+00,   5.55111512e-17,   2.22044605e-16],
       [  0.00000000e+00,   1.00000000e+00,   5.55111512e-16],
       [  1.11022302e-16,   0.00000000e+00,   1.00000000e+00]])
```

It is also possible to generate inner, outer, and cross products of matrices and vectors. For vectors, note that the inner product is equivalent to the dot product:

```
>>> a = np.array([1, 4, 0], float)
>>> b = np.array([2, 2, 1], float)
```

```
>>> np.outer(a, b)
array([[ 2.,  2.,  1.],
       [ 8.,  8.,  4.],
       [ 0.,  0.,  0.]])
>>> np.inner(a, b)
10.0
>>> np.cross(a, b)
array([ 4., -1., -6.])
```

Basic Array Operations

- Other Statistics

The median can be found:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form [[x1, x2, …], [y1, y2, …], [z1, z2, …], …] where x, y, z are different observables and the numbers indicate the observation times:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.        ,  0.72870505],
       [ 0.72870505,  1.        ]])
```

Here the return array `c[i,j]` gives the correlation coefficient for the ith and jth observables. Similarly, the covariance for data can be found:

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

Basic Array Operations

- Random Numbers: using the sub-module *random*

An array of random numbers in the half-open interval [0.0, 1.0) can be generated:

```
>>> np.random.rand(5)
array([ 0.40783762,  0.7550402 ,  0.00919317,  0.01713451,  0.95299583])
```

The `rand` function can be used to generate two-dimensional random arrays, or the `resize` function could be employed here:

```
>>> np.random.rand(2,3)
array([[ 0.50431753,  0.48272463,  0.45811345],
       [ 0.18209476,  0.48631022,  0.49590404]])
>>> np.random.rand(6).reshape((2,3))
array([[ 0.72915152,  0.59423848,  0.25644881],
       [ 0.75965311,  0.52151819,  0.60084796]])
```

To generate a single random number in [0.0, 1.0),

```
>>> np.random.random()
0.70110427435769551
```

To generate random integers in the range [min, max) use `randint(min, max)`:

```
>>> np.random.randint(5, 10)
9
```

SciPy

- NumPy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays.

- SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

- SciPy provides some basic functions to work with images. For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images.

SciPy

- NumPy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays.

- SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

- SciPy provides some basic functions to work with images. For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images.

SciPy Example

```
from scipy.misc import imread, imsave, imresize

# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

Left: The original image. Right: The tinted and resized image.

Further References

- Chapter 2 of [T1]